Title	A Dynamic Reconfiguration Tolerant Self-stabilizing Token Circulation Algorithm in Ad-Hoc Networks (Evolutionary Advancement in Fundamental Theories of Computer Science)
Author(s)	Kakugawa, Hirotsugu; Yamashita, Masafumi
Citation	数理解析研究所講究録 (2004), 1375: 274-281
Issue Date	2004-05
URL	http://hdl.handle.net/2433/25608
Right	
Type	Departmental Bulletin Paper
Textversion	publisher

数理解析研究所講究録 1375 巻 2004 年 274-281

A Dynamic Reconfiguration Tolerant Self-stabilizing Token Circulation Algorithm in Ad-Hoc Networks

広島大学　角川裕次
Hirotsugu Kakugawa
Hiroshima University, Japan
h.kakugawa@computer.org

九州大学　山下雅史
Masafumi Yamashita
Kyushu University, Japan
mak@csce.kyushu-u.ac.jp

Abstract

In this paper, we propose a self-stabilizing token circulation algorithm in ad-hoc networks. We propose a concept of dynamic reconfiguration tolerant self-stabilization as an extension of self-stabilization for discussing correctness of distributed algorithms that allows dynamic change of network topology. Intuitively, starting from any initial configuration, a dynamic reconfiguration tolerant self-stabilizing algorithm guarantees a property P, as long as the network topology dynamically changes within a constant C. (The definition is the same as the one of self-stabilization, if we define C as "the network is stable".)

The token visits processes in depth-first search manner along a spanning tree. If a network configuration is stable, our algorithm obtains a minimum spanning tree in at most $2n(n-1)$ steps, and the token visits processes along the minimum spanning tree.

The proposed algorithm guarantees that the token visits each node at least once within every $6(n-1)$ steps, if the interval of edge disconnections is at least $6(n-1)$ steps.

1 Introduction

Ad-hoc networks consist of mobile terminals with wireless communication devices. There is no pre-existing infrastructure for communication, and a terminal is connectable to an ad-hoc network without configuring it. This is a fascinating feature for end-users, but is a seed of the following technical difficulties, when to implement applications in ad-hoc networks: (1) since there are no access points that route messages among mobile terminals, the mobile terminals must route messages by themselves; (2) since the network topology rapidly changes as "mobile" terminals migrate, communication protocols must be adaptive to dynamic changes of topology; (3) since a terminal may join or even leave the ad-hoc network while participating in an application job, communication protocols must be robust against communication faults such as a network partition.

A *self-stabilizing* system is a non-masking fault tolerant distributed system such that it tolerates any finite number of transient faults [2, 6, 3]. There have been proposed many self-stabilizing algorithms, but most of them assume that the network topology be static. In fact, it is easy to observe that some constraint is necessary to introduce on the possible behavior of a network for an algorithm to achieve a meaningful task.

1.1 Our contribution

Whether or not a self-stabilizing algorithm for an ad-hoc network is correct depends on how the network dynamically changes. For a given constraint C on dynamic change of the network, we hence say that a system is *dynamic reconfiguration tolerant* (DRT for short) self-stabilizing (under C), if the system works as a correct self-stabilizing system as long as the network change does not violate C.

Under some moderate network reconfiguration constraint C, we propose a *stateless DRT self-stabilizing token circulation algorithm* for ad-hoc networks. Unlike the algorithm by Chen and Welch [1], our network change constraint C allows network to change while the system is converging. Unlike algorithms based on random walks, our algorithm deterministically circulates a token along a spanning tree, and hence the worst case waiting time for the token is deterministically bounded by $O(n)$. You may wish to assign to each edge a transmission cost. If the network remains static, the token is eventually circulated along a minimum spanning tree. The circulation cost is thus at most twice as much as the optimal cost (i.e., the solution to the traveling salesman problem) when the cost satisfies the triangle inequality.

Another advantage of our algorithm is a short convergence time. Unlike the random walk approach, we do not make use of a collision-and-elimination scheme to remove redundant tokens. Instead, an initiator issues a priority to a token, and the elimination is done by an initiator based on the priority. The convergence time to eliminate unnecessary tokens is deterministically bounded by $O(n)$, and the spanning tree along which the token is circulated converges to a minimum spanning tree in $O(n^2)$-time, if the network remains static. There have been proposed some algorithms for constructing the minimum spanning tree [4, 5], but they are not applicable for our purpose of designing a stateless protocol.

Token circulation algorithms based on random walk are in general space efficient. Our algorithm

uses $O(n \log n)$ bits for a token, and a variable of constant size at each initiator, but no memory is necessary for a non-initiator, which is more space efficient than the algorithm by Chen and Welch [1].

This paper is organized as follows: Section 2 presents a computation model of distributed system. In Section 3, we propose a stateless self-stabilizing token circulation algorithm in ad-hoc networks, and then prove its correctness in Section 4. In Section 5, we give concluding remarks.

2 The Model

2.1 Network Model

We consider a system consisting of n mobile terminals with wireless communication devices. We model such a system by a set of processes $V = \{p_1, p_2, ..., p_n\}$ with unique identifiers. For each process p_i, let N_i be the set of neighbor processes that p_i can *directly* communicate with. We assume that every communication channel is bidirectional; $p_j \in N_i$ if and only if $p_i \in N_j$. N_i's for all $p_i \in V$ define a network $G = (V,E)$, where $(p_i, p_j) \in E$ if and only if $p_j \in N_i$.

To each edge $(p_i, p_j) \in E$, we assign a positive *weight* (or cost) denoted by $w_{i,j} (= w_{j,i})$. The weight $w_{i,j}$ may also dynamically change. We can however assume that the weights $w_{i,j}$ are unique without loss of generality, since otherwise, we can use triples $(w_{i,j}, p, q)$ as unique weights instead, where $p = \min\{p_i, p_j\}$ and $q = \max\{p_i, p_j\}$. The minimum spanning tree is thus uniquely determined.

Since terminals may change their locations, N_i may dynamically change and so may G accordingly. We however assume that the values of N_i and $w_{i,j}$ for all $p_j \in N_i$ are correctly maintained. A crucial assumption is that they never change while a token is visiting p_i.

Our network is synchronous in the sense that 1) all local clocks show the same time, and 2) there is an upper bound δ on the communication delay between two neighboring processes, where δ is available for the processes. Without loss of generality, we assume that $\delta = 1$ (unit time) in the rest of this paper.

Changes of network topology may occur in our model. Suppose that the system is partitioned into several sub-networks and this situation continues forever. Then all what a process in a sub-network can hope is to circulate a token among the processes in the sub-network. A change of the network topology may be viewed from the process as a join or a leave of another process to or from the (sub-)network. We thus consider V as the set of all processes that have chances to participate in the system, and assume that the size $|V| = n$ is also available for the processes.

We discuss *stateless* algorithms in the sense that *non-initiators* do not need to maintain local variables. However that non-initiators need to temporarily use local variables to process a token. We assume that processes allocate memory to the local variables when a token arrives, and release them when the token leaves.

2.2 Dynamic reconfiguration tolerant self-stabilization

We now define dynamic reconfiguration tolerant self-stabilizing systems.

Definition 1 *A system is a* dynamic reconfiguration tolerant *(DRT for short) self-stabilizing system with respect to a specification P under a dynamic network reconfiguration constraint C if the following conditions are satisfied.*

1. *Convergence: For any initial configuration and for any computation starting from it, the system eventually satisfies P, if network configuration changes follow C.*
2. *Safety: For any initial configuration that satisfies P and for any computation starting from it, the system remains to satisfy P, as long as network configuration changes follow C.*

If we adopt a constraint "no transient error and network reconfiguration occur" for C, a DRT self-stabilizing system with respect to P under C is obviously a conventional self-stabilizing system with respect to P. Let I, C, S and L be an initial condition, a dynamic network reconfiguration constraint, a (safety) property and a (liveness) property, respectively.

Definition 2 *A system is said to be* (I,C,S)-safe *if S is always true for any computation starting with an initial configuration that satisfies I, as long as network configuration changes follow C.*

Definition 3 *A system is said to be* (I,C,L)-live *if L becomes true eventually for any computation starting with an initial configuration that satisfies I, provided that network configuration changes follow C.*

These two concepts will play central roles in the correctness proofs of our algorithms. Some primitive properties of (I,C,P)-safety and (I,C,P)-liveness are summarized below.

- If a protocol A is (I_1,C,S)-safe and (I_2,C,S)-safe, then A is $(I_1 \vee I_2,C,S)$-safe.
- If a protocol A is (I,C,S_1)-safe and (I,C,S_2)-safe, then A is $(I,C,S_1 \wedge S_2)$-safe.
- If a protocol A is (I,C,S)-safe, then A is $(I \wedge I',C,S)$-safe for any I', and $(I,C \wedge C',S)$-safe for any C'.
- If a protocol A is (I_1,C,S_1)-safe and (S_1,C,S_2)-safe, then A is (I_1,C,S_2)-safe.
- If a protocol A is (I_1,C,L)-live and (I_2,C,L)-live, then A is $(I_1 \vee I_2,C,L)$-live.
- If a protocol A is (I,C,L_1)-live and (I,C,L_2)-live, then A is $(I,C,L_1 \wedge L_2)$-live.
- If a protocol A is (I,C,L)-live, then A is $(I \wedge I',C,L)$-live for any I', and $(I,C \wedge C',L)$-live for any C'.
- If a protocol A is (I_1,C,L_1)-live and (L_1,C,L_2)-live, then A is (I_1,C,L_2)-live.

3 The Algorithm

This section presents a stateless and DRT self-stabilizing token circulation algorithm that circulates a token along the minimum spanning tree edges in an ad-hoc network.

3.1 Overview

A process who is interested in token circulation becomes an initiator. Hence more than one process may become an initiator. The algorithm consists of two threads, one for an initiator and the other for all processes, including *initiators*, who receive a token. Definition of the token is shown in Figure 1.

These two threads are hence executed as two threads in a single process of the initiator.

1. *Initiator thread:* This thread, whose code is given in Figure 2, is executed by an initiator p_i. Process p_i continues executing this thread as long as it is interested in the token circulation.
2. *Token thread:* This thread is executed by any process p_i who receives a token. The code is shown in Figure 4, with a macro defined in Figure 3. This thread immediately terminates when the token is sent to a neighbor process or is discarded.

In the descriptions of code, we use Java-like *try-catch* constructs to describe an exception for time-out error and signal handling. Because only initiators maintain local states, we can consider the pair of the token and the code for token thread as an *agent* that travels processes in a network. By $p.m$, we denote the local variable m at an initiator process p.

We would like to give readers a rough idea to maintain the minimum spanning tree using a stateless algorithm. Other features will be discussed later.

An initiator process generates a token and forward it to its neighbor. As part of its information, the token carries a tree that spans the processes it has visited in terms of the set of tree edges. When the token is initialized by an initiator, the token carries an empty edge set, i.e., empty tree. The token is sent by a process p_i in a depth-first graph search manner to its neighbor p_j. When p_j receives the token for the first time, it updates the tree edge set in the token by adding an edge (p_i, p_j). Note that p_j is selected so that this addition does not create a cycle. After a while, the token will carry a spanning tree edge set T, which however may not be the minimum spanning tree.

After a spanning tree is constructed, whenever the token returns to its initiator, the initiator process improves the weight of the spanning tree by replacing an edge in the tree with a non-tree edge which improves the weight of the tree. Such a non-tree edge is searched during the last traversal of the tree. We improve the cost of a spanning tree by replacing one edge per circulation, and the spanning tree eventually becomes minimum.

One may think that more edges should be replaced in a circulation. Unfortunately, such a scheme does not guarantee that the token come back to the initiator within a reasonable interval. That is why updating a spanning tree during a circulation changes the route the token visits. As a result, the initiator must choose larger timeout value for regenerating a token to cope with token loss. This makes recovery from token loss slow. Although such a worst case is unlikely to happen in practice, we give theoretical guarantee in this paper.

3.2 Token structure

Since our algorithm is stateless, a token carries all data necessary for circulation, including the current spanning tree. The data structure of a token is given in Figure 1. Let t be a token.

- $t.tree$: The set of ordered edges that represents a rooted (ordered) tree, along which t is circulated.
- $t.type \in \{\mathsf{probe}, \mathsf{echo}\}$: The direction of traversal. Token t is being sent toward a leaf when $t.type = \mathsf{probe}$, and is being echoed back to the root when $t.type = \mathsf{echo}$.
- $t.wgt$: The weights of edges in $t.tree$.
- $t.age$: The age of t, whose value is initially 0 and is incremented by one, whenever t is sent from a process to another. The age is reset to 0 when the token returns to its initiator. Thus a token whose age goes beyond some threshold value can be eliminated, since its initiator would have already left the network.
- $t.ini$: The identifier of initiator.
- $t.id$: The identifier of t assigned by the initiator selected from an integer set $\{0, 1, \ldots, M-1\}$, where we assume that M is large enough so that more than M tokens never exist in the network at a time.[1]
- $t.alte$: The edge e is a candidate edge to improve the weight of $t.tree$ such that 1) e has found in this traversal, 2) $e \notin t.tree$, and 3) the unique cycle in $t.tree \cup \{e\}$ contains an edge with a weight larger than e's. The value is reset to $\perp$ when the initiator starts a new round of circulation.
- $t.altw$: The weight of $t.alte$.

3.3 Network parameters and functions

The codes for p_i use the following network parameters and functions. Network parameters are as follows:

- N_i : Set of current neighbors of p_i.
- $w_{i,j}$: Current weight of edge (p_i, p_j).

Note that N_i and $w_{i,j}$ may autonomously change their values during the execution. Let $p_j, p_k \in N_i$ be two neighbors of p_i. Then we say that p_j is *smaller* than p_k if $w_{i,j} < w_{i,k}$ in the following.

The functions are as follows:

- $Procs(t)$: The process set of $t.tree$, i.e., the set of processes that t has visited.
- $Root(t)$: The root of $t.tree$.
- $Parent(t, p_i)$: The parent of p_i in $t.tree$. If p_i is the root then $Parent(t, p_i) = \perp$.

[1] As will be clear from the algorithm given in section 3, this assumption is removable and M can be set any value ≥ 2, at the expense of the convergence time; the algorithm guarantees that the number of tokens is reduced to at least $1/M$th in every τ ticks, once the network becomes stable.

Message format of token : $\langle type, tree, wgt, age, id, ini, alte, altw \rangle$

type :	{ probe, echo }	
tree :	**set of pairs** ⟨**process identifier, process identifier**⟩	— *Directed edges of a spanning tree.*
wgt :	**set of triples** ⟨**edge weight, process identifier, process identifier**⟩	— *Weights of the directed edges.*
age :	**integer**	— *Number of traversed edges in the current circulation.*
id :	**integer**	— *Token identifier.*
ini :	**process identifier**	— *Identifier of the initiator.*
alte :	⟨**process identifier, process identifier**⟩ or $\perp$	— *Candidate for the minimum spanning tree edges.*
altw :	**edge weight**	— *Weight of alte.*

Figure 1: The data structure of a token.

- *Children*(t, p_i): The set of children of p_i in $t.tree$.
- *FirstChild*(t, p_i): The smallest child of p_i in $t.tree$. If p_i has no children in $t.tree$, then *FirstChild*$(t, p_i) = \perp$.
- *NextChild*(t, p_i, p_j): The smallest child among those of p_i in $t.tree$ larger than p_j.
- *TreeNeighbors*(t, p_i): A set of neighbor processes of p_i in $t.tree$. By definition, we have *TreeNeighbors*(t, p_i) = *Parent*$(t, p_i) \cup$ *Children*(t, p_i).
- *Alive*(t): The predicate that returns **true** if and only if $t.age \leq \alpha$. The age is the number of edges that t has traversed after visiting the initiator for the last time.

  The value of α is discussed later, depending on the degree of dynamic change of network topology.

3.4 Token behavior

We explain the behavior of the token as if the token is a *mobile agent* that travels nodes in the network. (Below, we use the term "node" and "process" interchangeably.) The code for the token is shown in Figure 4.

A token moves in nodes in a depth-first fashion. It never gives up traveling nodes even if the network topology dynamically changes. In such a case, a token looks for a new route based on the local view of the topology. Exceptional events that a token gives up traveling (i.e., token is discarded) are (1) the existence of an initiator process with higher priority, or (2) expiration of lifetime of a token.

Initially a tree in the token is empty, and it grows as it visits a new process. The token is typed probe (resp. echo) if the token is forwarded from a parent to a child (resp. from a child to a parent).

As described above, suppose that an initiator p^* creates a new token t with $t.type$ = probe, and the token is sent to the smallest neighbor (line 8–11 of Figure 2).

4 Correctness and Performance

First, we discuss general properties of the proposed algorithm.

Theorem 1 *The length of a token is $O(n \log n)$ bits, where n is the number of processes.* □

Lemma 1 *For any initial configuration in which no token exist in the network, eventually at least one token is generated.* □

Regardless any dynamic change of network configuration, a token t is discarded only when (1) it arrives at an initiator process whose priority is higher than that of t, or (2) its age reaches its lifetime. This is true even if edge weights and network topology dynamically changes. We formally state this fact in terms of (I, C, S)-safety and (I, C, L)-liveness as follows.

- I_1^S : There is only one token t in the network, $p^*.m = t.id$, and t is generated by an initiator p^* whose priority is the highest among all initiators.
- C_1^S :
  - A set of neighbor processes N_i never change when t is visiting at p_i for each p_i
- S_1^S : Token t remains to exist if its age is less than the lifetime and timeout time of p^* expires.

Note: The constraint C_1^S implies that N_i never becomes empty since it includes p_j which is the previous process t visited.

Lemma 2 *The proposed protocol is (I_1^S, C_1^S, S_1^S)-safe.* □

4.1 Dynamic network without edge disconnections

First, we consider a case that there is only one initiator in the network. Later, we discuss the case that there are more than one initiator in the network.

We define I_1^D, C_1^D and L_1^D as follows.

- I_1^D :
  - There is an initiator p^* in the network, whose priority is the highest among all initiators,
  - There is only one token t in the network generated by p^* such that $p^*.m = t.id$,
  - $t.age + 4(n-1) \leq \alpha$ holds, where n is the number of processes.
- C_1^D :
  - A set of neighbor processes N_i never change when t is visiting at p_i for each p_i.
  - No new initiator process whose priority is higher than that of p^* appear,

```
Variables of an initiator p_i :
  m : integer initially 0;   — Token identifier.

Code for an initiator p_i :
1:  while {   — initiate new circulation by generating a new token.
2:    try {
3:      wait;   — Wait for any token to arrive (with timeout). Token is handled by the token thread.
4:    } catch (Signal) {   — A token visits this process. This event is nortified by the token thread.
5:      ;   — Do nothing. Wait for next arrival of a token.
6:    } catch (TimeoutException) {
7:      m := (m+1) mod M;   — Assign new token identifier.
8:      t := ⟨probe, ∅, ∅, 0, m, p_i, ⊥, ∞⟩
9:      Let p_k be a process in N_i such that w_{i,k} is the smallest;
10:     send t to p_k;
11:   }
12: }
```

Figure 2: Initiator thread: code for an initiator.

- Edge disconnections never happen,
- There is always a connected path in the network from a process t locates to p^*,
- There may be any number of dynamic edge additions and weight changes, and
- The number of connected processes in the network never increases.

- L_1^D : Token t eventually returns to p^*.

Note: I_1^D represents an initial configuration just after network topology is changed, such as edge disconnections. Thus, the initial value of $t.tree$ may not be a correct tree. Intuitively, the (I_1^D, C_1^D, L_1^D)-liveness states that the token visits p^* within $4(n-1)$ edge traversals, even if edge weights and edge additions dynamically occur.

Lemma 3 *The proposed algorithm is (I_1^D, C_1^D, L_1^D)-live, and The token returns to p^* within $4(n-1)$ edge traversals.* □

Consider when t arrives the initiator p^*, as discussed in the previous lemma. In case the root of $t.tree$ is not p^*, which implies that p^* was deleted from $t.tree$, p^* resets t and starts the next round.

In case the root of $t.tree$ is p^*, the tree held in $t.tree$ may still contain processes and edges that do not exist. This is why t may have not visited all the processes before it arrives at p^*.

Therefore, one more round is required so that t can visit all the processes to hold a spanning tree in $t.tree$. Below, we consider the behavior of the algorithm when t visits p^*.

- I_2^D :
  - There is an initiator p^* in the network, whose priority is the highest among all initiators,
  - There is only one token t in the network generated by p^* such that $p^*.m = t.id$,
  - Token t locates at p^*, and
  - $t.age = 0$.
- $C_2^D = C_1^D$
- L_2^D : The token t returns to p^* with a spanning tree in $t.tree$, and each process is visited at least once.

Lemma 4 *Assume that $\alpha \geq 2(n-1)$ and $\tau \geq 2(n-1)$. Then, the proposed algorithm is (I_2^D, C_2^D, L_2^D)-live. The token returns to p^* within two rounds, each of which requires at most $2(n-1)$ edge traversals.* □

Above lemma makes clear the reason why we do not improve a spanning tree during a round. If we replace tree edges during a round, there is no guarantee to return the initiator within $2(n-1)$.

Even if the network topology is stable, edge weights are likely to change. Next, we show a property of the proposed algorithm under such change of network configuration.

- I_3^D :
  - There is an initiator p^* in the network, whose priority is the highest among all initiators,
  - There is only one token t in the network generated by p^* such that $p^*.m = t.id$,
  - Token t locates at p^*,
  - $t.age = 0$. and
  - $t.tree$ contains a spanning tree.
- $C_3^D = C_2^D (= C_1^D)$
- L_3^D : The token t returns to p^* with a spanning tree in $t.tree$ and each process is visited at least once.

Lemma 5 *Assume that $\alpha \geq 2(n-1)$ and $\tau \geq 2(n-1)$. Then, the proposed algorithm is (I_3^D, C_3^D, L_3^D)-live. The token returns to p^* within $2(n-1)$ edge traversals.* □

Lemma 6 *Assume that $\alpha \geq 2(n-1)$ and $\tau \geq 2(n-1)$. Then the proposed algorithm is $(I_2^D, C_2^D, I_2^D \wedge L_2^D)$-live and $(I_3^D, C_3^D, I_3^D \wedge L_3^D)$-live.* □

We define a safety property S_3^D as follows.

- S_3^D : There exist only one token t in the network which contains a spanning tree in $t.tree$.

We have the following theorem.

```
1:  macro UpdateToken ≡
2:  {
3:      // IMPROVE A SPANNING TREE.
4:      if (t.alte ≠ ⊥) {   — There is an edge to improve the spanning tree.
5:          t.tree := t.tree ∪ {t.alte};   — Temporarily t.tree has a cycle.
6:          Find an edge e in t.tree such that
7:              t.tree − {e} is a spanning tree and its weight is the smallest;
8:          t.tree := t.tree − {e};
9:          Delete from t.wgt the weight of edge e;
10:         Add into t.wgt edge t.alte with weight t.altw;
11:     }
12:     // REFRESH THE TOKEN FOR THE NEXT ROUND OF TOKEN CIRCULATION.
13:     if (p_i = t.ini) {
14:         m := (m + 1) mod M;
15:         t.id := m;   — Assign new token identifier.
16:         t.age := 0;   — Reset token age.
17:     }   — If p_i (= the root of t.tree) is not the initiator of t, t.age and t.id are unchanged.
18:     t := ⟨probe, t.tree, t.wgt, t.age, t.id, t.ini, ⊥, ∞⟩;   — Assign new token identifier and reset the token age.
19: }

20: macro FindCandidate ≡
21: {
22:     if (t.alte = ⊥)
23:         Let T be t.tree;
24:     else
25:         Let T be a spanning tree with the smallest weight among subgraphs of t.tree ∪ t.alte;
26:     Let T' be a spanning tree with the smallest weight among subgraphs of T ∪ {(p_i, p_ℓ) : p_ℓ ∈ N_i − TreeNeighbors(t)};
27:     if (weight of T' < weight of T){
28:         Let p_ℓ be a process that yields T';
29:         return p_ℓ;
30:     }
31:     return ⊥;
32: }
```

Figure 3: Macro definition for token thread.

Theorem 2 *Assume that* $\alpha \geq 2(n-1)$ *and* $\tau \geq 2(n-1)$. *Then, the proposed algorithm is dynamic reconfiguration tolerant self-stabilizing with respect to a specification* S_3^D *under a dynamic network reconfiguration constraint* C_1^D. □

Corollary 1 *Assume that* $\alpha \geq 2(n-1)$ *and* $\tau \geq 2(n-1)$. *Suppose that an initial configuration satisfies* I_3^D *and network dynamically changes with a constraint* C_3^D. *Then, for each process* p_i, *the interval that a token visits* p_i *is at most* $4(n-1)$, *if the timeout value* τ *at initiators is at least* $4(n-1)$. □

Note that *t.tree* may not become a minimum spanning tree if edge weights change dynamically.

4.2 Dynamic network with edge disconnections

In the previous subsection, we considered a case just after edge disconnection happen and no more edge disconnection happen thereafter. Now we consider a case that edge disconnection dynamically occur.

We define C_4^D and S_4^D as follows. Note that C_4^D differs from $C_1^D (= C_2^D = C_3^D)$ only in the condition of edge disconnections.

- C_4^D :
  - A set of neighbor processes N_i never change when t is visiting at p_i for each p_i.
  - No new initiator process whose priority is higher than that of p^* appear,
  - Any number of edge disconnections may happen simultaneously, provided that the time interval between such events is at least $6(n-1)$,
  - There is always a connected path in the network from a process t locates to p^*,
  - There may be any number of dynamic edge additions and weight changes, and
  - The number of connected processes in the network never increases.
- S_4^D :
  - Only one token t exists (and it remains to exist).

Theorem 3 *Assume that* $\alpha \geq 6(n-1)$ *and* $\tau \geq 6(n-1)$. *Then, the proposed algorithm is dynamic reconfiguration tolerant self-stabilizing with respect to a specification* S_4^D *under a dynamic network reconfiguration constraint* C_4^D. *The token visits each process with interval at most* $6(n-1)$. □

Next, we consider a case that edge disconnections happen more frequently. We define C_5^D and S_4^D as follows.

```
When a token t arrives at p_i from p_j :
 1:  t := receive;
 2:  t.age := t.age + 1;   — Increment the age by one.
 3:  if (p_i is an initiator) ∧ ((t.ini > p_i) ∨ ((t.ini = p_i) ∧ (t.id ≠ m))) {   — Discard the token based on priority.
 4:      Discard t;
 5:  } else if (¬Alive(t))   — The token is too old to alive.
 6:      Discard t.
 7:  } else {
 8:      if (p_i is an initiator)
 9:          signal;   — Restart time-out timer of the initiator thread.
10:      // EXTEND THE SPANING TREE IF p_i IS NOT INCLUDED YET.
11:      if (t.type = probe) ∧ ((p_j, p_i) ∉ t.tree){   — This is the first visit to p_i.
12:          t.tree := t.tree ∪ {(p_j, p_i)}; t.wgt := t.wgt ∪ {(p_j, p_i, w_i(p_j))};   — Extend the spanning tree.
13:          t.alte = ⊥; t.altw = ∞;   — Reset the candidate for improving the spanning tree.
14:          if (t.ini = p_i)   — The token visits an initiator p_i which was disconnected from t.tree.
15:              t := ⟨probe, ∅, 0, m, p_i, ⊥, ⊥, ⊥, ∞⟩   — Reset t and start new round.
16:      }
17:      // CHECK IF NETWORK TOPOLOGY AND EDGE WEIGHTS ARE CHANGED.
18:      for each p_k ∈ (Children(t, p_i) − N_i)   — A child p_k is disconnected from p_i.
19:          Delete a subtree rooted by p_k from t.tree, and update t.wgt accordingly;
20:      if (Parent(t, p_i) ∉ N_i)   — Parent process is disconnected from p_i.
21:          t.tree := a subtree of t.tree rooted by p_k, and update t.wgt accordingly;
22:      for each p_k ∈ TreeNeighbors(t, p_i) {
23:          if (the weight of (p_k, p_i) in t.wgt is different from w_i(p_k))
24:              Update the weight of (p_k, p_i) in t.wgt to be w_i(p_k);
25:      }
26:      // FIND A CANDIDATE EDGE TO IMPROVE THE SPANNING TREE.
27:      if (N_i − Procs(t) = ∅) {
28:          p_ℓ := FindCandidate;   — p_ℓ is in N_i − TreeNeighbors(t) or equals ⊥. (See Figure3.)
29:          if (p_ℓ ≠ ⊥) {   — Better candidate is found.
30:              t.alte = (p_i, p_ℓ); t.altw = w_i(p_ℓ);
31:          }
32:      }
33:      // FIND A DESTINATION OF THE TOKEN.
34:      if (N_i − Procs(t) ≠ ∅) {   — There is a neighbor process not in the spanning tree.
35:          t.type := probe; p_k := a process such that w_i(p_k) is the smallest among p_k ∈ N_i − Procs(t);
36:      } else if (p_i is a leaf process of t.tree){
37:          t.type := echo; p_k := Parent(t, p_i);   — Send t back to the parent (p_k = p_j).
38:      } else {
39:          if (t.type = probe) {   — The token is sent from the parent.
40:              p_k := FirstChild(t, p_i);   — Forward the token to the first child.
41:          } else {   — The token is sent back from a child (t.type = echo).
42:              p_k := NextChild(t, p_i, p_j);   — Forward the token to the next child.
43:          }
44:          if (p_k ≠ ⊥) {   — There is next child to forward.
45:              t.type := probe;   — Forward the token of probe type to the child.
46:          } else {   — No more next child to forward (p_k = ⊥).
47:              if (p_i = Root(t)) {
48:                  UpdateToken;   — The end of a round. Improve the spanning tree, and prepare for the next round. (See Figure3.)
49:                  p_k := FirstChild(t, p_i);   — Forward token t to the first child.
50:              } else {
51:                  p_k := Parent(t, p_i);   — Non-root sends the token back to its parent.
52:              }
53:          }
54:      }
55:      // FORWARD THE TOKEN.
56:      send t to p_k;
57:  }
```

Figure 4: Token thread: code for a process who receives a token.

- C_5^D : the same as C_4^D, except the time interval between edge disconnection events is at least $4(n-1)$, and
- $S_5^D = S_4^D$

Theorem 4 *Assume that* $\alpha \geq 6(n-1)$ *and* $\tau \geq 6(n-1)$. *Then, the proposed algorithm is dynamic reconfiguration tolerant self-stabilizing with respect to a specification* S_5^D *under a dynamic network reconfiguration constraint* C_5^D. □

Note that the value $6(n-1)$ is the minimum interval of edge disconnections so that the token visits each process. There are many patterns for edge disconnections with interval less than $6(n-1)$ that guarantees the safety property S_4^D.

Observation 1 *Theorem 3 holds even if any number of edge disconnections happen at any time provided that each disconnected edge is not a tree edge.*

4.3 Stable network

Next, we consider that the network is stable. In this case, the token eventually computes a minimum spanning tree, and it travels processes along a minimum spanning tree.

We define I_1^S, C_1^S and L_1^S as follows. Because stable network is a special case of dynamic network, we assume that the liveness property L_3^D holds in the initial configuration.

- I_1^S :
  - There is an initiator p^* in the network, whose priority is the highest among all initiators,
  - There is only one token t in the network generated by p^* such that $p^*.m = t.id$,
  - Token t locates at p^*,
  - $t.age = 0$, and
  - $t.tree$ contains a spanning tree.
- C_1^S :
  - A set of neighbor processes N_i never change at each process and edge weights never change (i.e., network is stable), and
  - No new initiator process appear.
- L_1^S : Token t eventually obtains a minimum spanning tree in $t.tree$.

Theorem 5 *The proposed algorithm is* (I_3^D, C_3^D, L_3^D)-*live, and* L_3^D *becomes true within* $2(n-1)^2$ *edge traversals.* □

Once a spanning tree is computed, the token travels the same route in each round, as long as the network is stable. Thus, we have the following corollary.

Corollary 2 *Suppose that the network is stable and a token holds a minimum spanning tree is computed. Then the interval that a token visits a process is* $2(n-1)$. □

4.4 Process join

Consider when a new process p_i joins to the network. The token t visits one of neighbor process, say p_j, of p_i, it moves to p_j because p_i is not in $t.tree$. Thus, new process is eventually included in a spanning tree by simply adding an edge (p_i, p_j). Because the number of processes in a network increases by one, the token requires addition two edge traversals to travel along a spanning tree.

In case some processes may join to the network, the value of α (lifetime of a token) and τ (timeout value) must be changed such that n is the upper bound on the number of processes in the network.

5 Conclusion

In this paper, we proposed a concept of dynamic reconfiguration tolerant (DRT) self-stabilization as a theoretical framework of distributed algorithm for dynamic ad-hoc networks. Then, we proposed a deterministic and stateless DRT self-stabilizing token circulation algorithm for dynamic ad-hoc networks. Our algorithm computes a minimum spanning tree for less communication complexity. By our algorithm, the interval of the token visit for each process is $O(n)$, which is deterministically bounded. In addition, the timeout value of initiator is also $O(n)$ which implies that the recovery from token loss is fast. In contrast, token circulation by random walks, the interval of the token visit is not deterministically bounded, and recovery from token loss is slow, $O(n^3)$.

We believe that our framework can be used for distributed algorithms for dynamic ad-hoc networks. Design and verification of algorithms under the framework is left for future works.

References

[1] Yu Chen and Jennifer L. Welch. Self-stabilizing mutual exclusion using tokens in mobile ad hoc networks. In *Proceedings of DIALM*, 2002.

[2] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, November 1974.

[3] S. Dolev. *Self-stabilization*. The MIT Press, 2000.

[4] R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and Systems*, 5(1):66–77, January 1983.

[5] J. A. Garay, S. Kutten, and D. Peleg. A sublinear time distributed algorithm for minimum-weight spanning trees. *SIAM Journal on Computing*, 27(1):302–316, February 1998.

[6] F. C. Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys*, 35:43–48, 1996.